基于Go构建海量作业作业平台

袁帅 villager

bilibili/基础架构部/SRE/平台工程组/资深开发工程师

目录

第一部分

作业平台简介

日常的运维管理无时无刻的不在执行，不管是执行软件安装、还是配置和启动服务，甚至是一个长时间的停机维护，将这些操作抽象来表达统称为作业。

运维作业的重要性在于它可以帮助运维/SRE团队更好的管理和维护系统，**确保系统的稳定性和可靠性**。通过自动化运维作业，**可以减少人工干预，降低出错率**，提高工作效率和质量

作业平台的需求来源

命令自动化

依靠开源工具(ansible，saltstack)，对命令进行操作，无web管理界面，无精细化权限管控

场景固化不灵活

SRE有固化的运维操作场景，新增场景支持需要二次开发

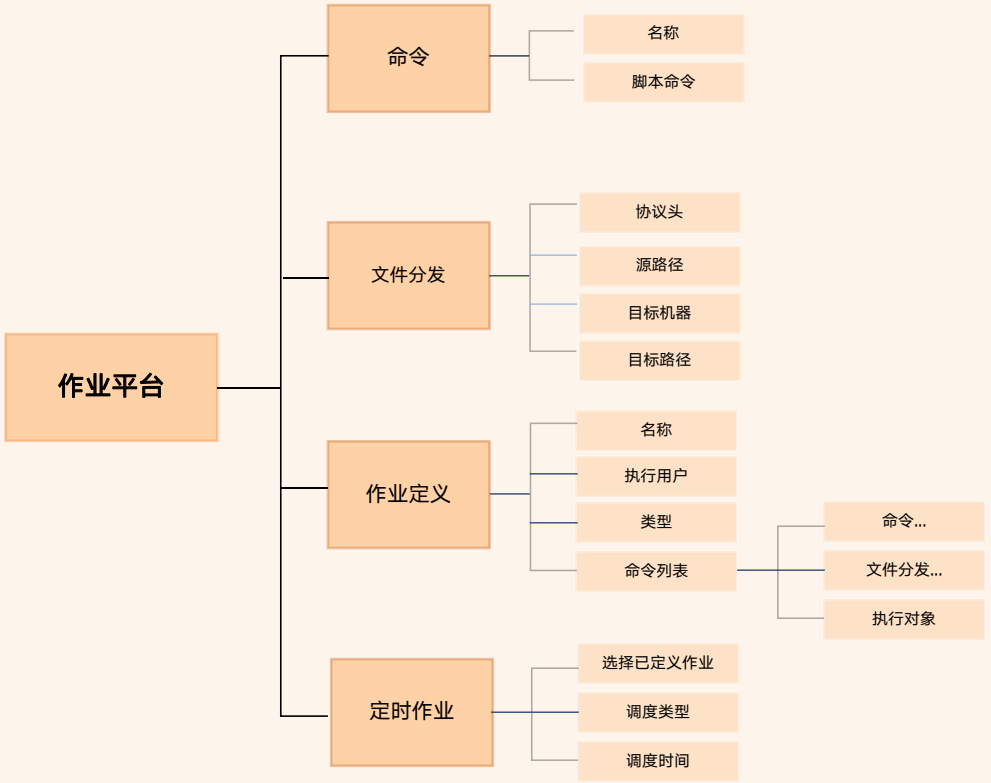
缺乏链路打通

自动化执行的主机对象和CMDB无法打通

定时任务泛滥

依靠Linux的crontab分散在各自主机上，无法统一管理

作业平台主要对象



作业平台主要对象

命令：一个可以独立操作，如：关机，重启等

文件分发/下载：将指定的文件分发到目标机路径;将目标机路径的文件下载到本地

作业：一系列命令、文件分发/下载的有序组合，它还包含执行对象

定时作业：定时执行的作业

第二部分

作业平台挑战

1. 系统集成：作业平台需要集成多个系统和工具，这些系统可能来自不同的操作系统或供应商
2. 自动化作业编排开发：运维作业平台的核心是自动化编排作业
3. 安全性：运维作业平台需要处理敏感数据和关键操作，平台需要采用安全的认证和授权机制
4. 性能优化：运维作业平台需要处理大量的数据和任务，因此需要具备高性能和可伸缩性
5. 故障排除：平台需要具备完善的故障排除机制和技术支持，以快速定位和解决问题
6. 用户体验：运维作业平台需要提供友好的用户界面和操作体验，以便用户轻松地管理和监控作业

作业平台挑战

安全挑战

更大的攻击面，网络/组件层面
数据安全性

多云集成困难

不同云直接连接编排

资源权限问题

资源变更影响，资源的权限管理

兼容性

跨越多个基础设施的兼容性是一个
棘手问题

回源问题

海外业务，作业日志，指标采集等
回源IDC同样棘手

运维复杂

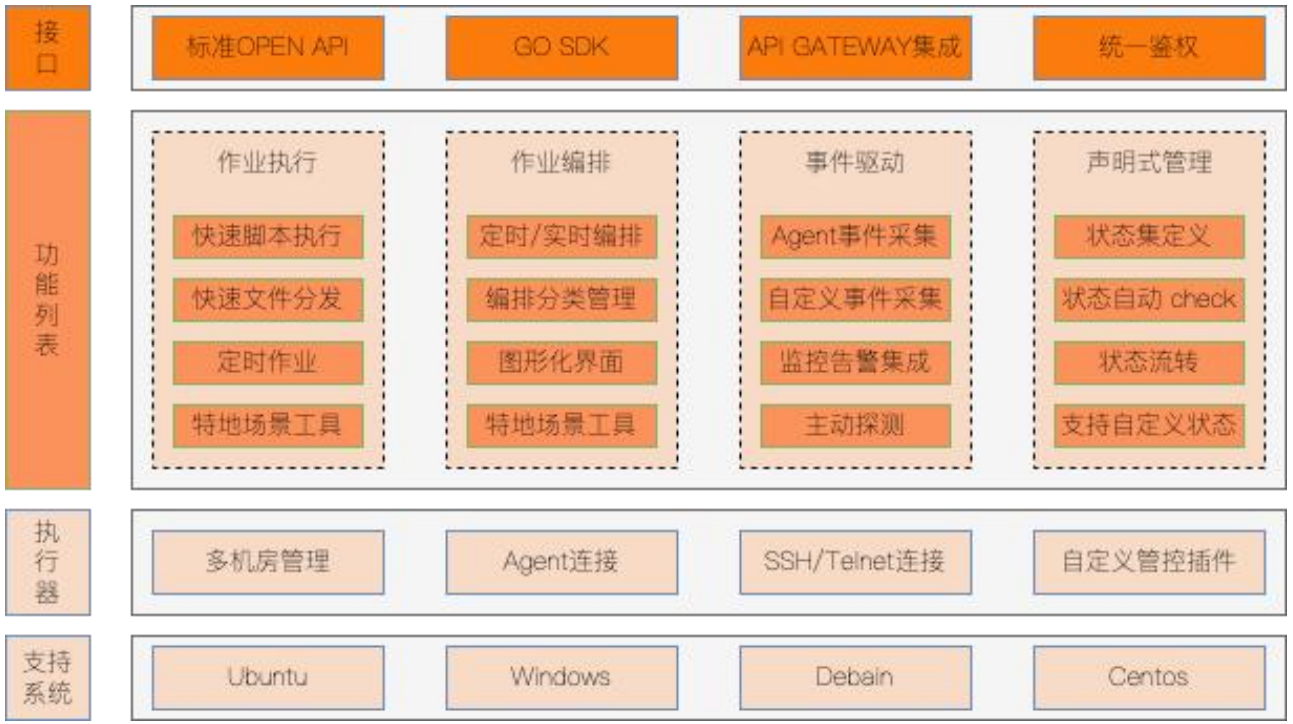
作业平台组件升级维护成本太高

第三部分

B站作业平台Job介绍

作业平台（Job）是一套基于作业平台Agent/SSH双模式，提供基础操作的原子平台；具备上万台机器并发处理能力，除了支持脚本执行、文件分发、定时任务等一系列基础运维场景以外，还支持通过流程调度能力将零碎的单个任务组装成一个自动化作业流程；而每个任务都可做为一个原子节点，提供给上层或周边系统/平台使用，实现调度自动化。

B站作业平台介绍



B站作业平台是基于管控平台管道基础之上，为用户提供原子操作的平台。支持脚本执行、文件分发、定时任务等一系列运维场景，同时支持将单个任务组装成一个自动化的作业流程。

1. 作业平台开放了 API 接口，以原子节点的形态开放提供给其它系统或平台进行调度
2. 作业分为作业模板和执行实例，两者是一对多的关系
3. B 站作业平台支持原生SSH 协议，扩展 Agent 模式
4. 跨 OS 操作系统支持，同时针对高危命令实时检测并阻拦

B站作业平台Job介绍

2023

多机房作业架构

多机房部署，作业平台进行多机房作业调度，从而实现机房多活

2021.02

基于 Agent/SSH双模式混合架构

相对独立的Agent，大大解决了机房内海量运维作业并发执行的场景

2018

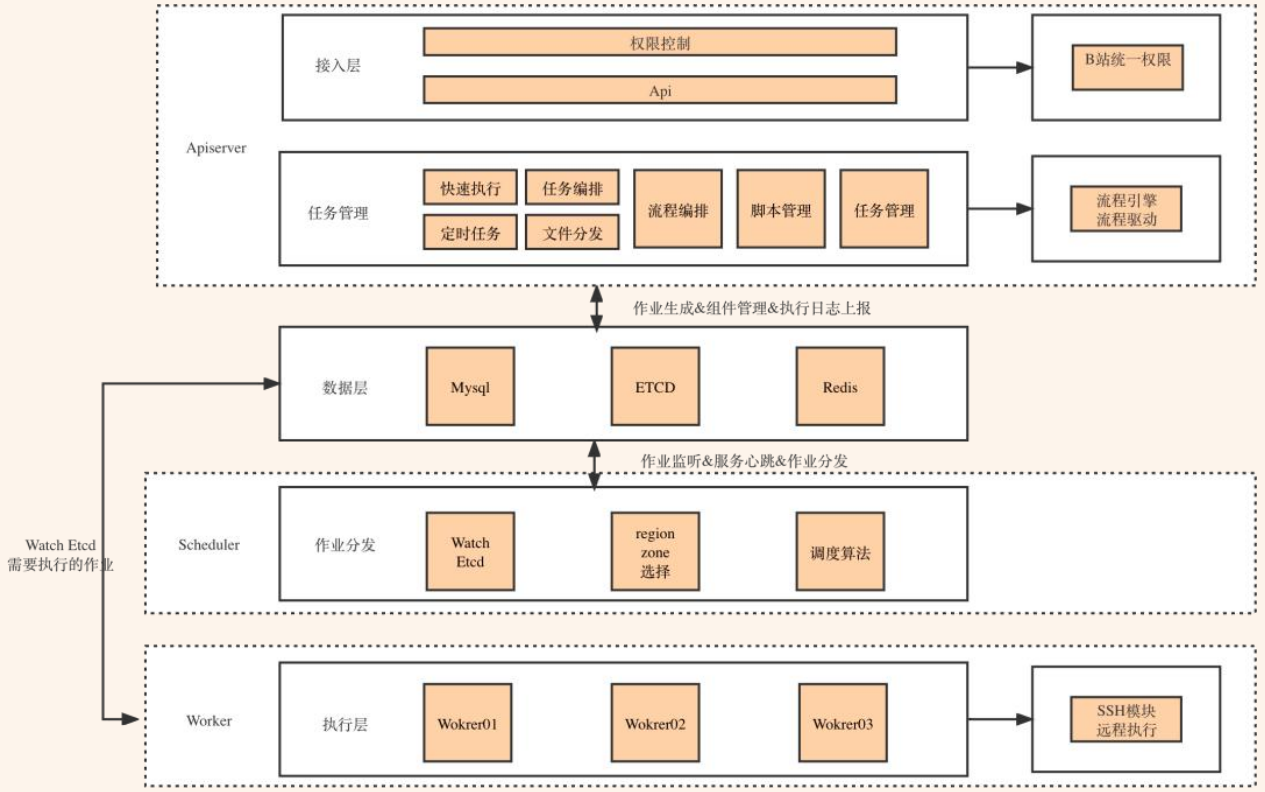
基于SSH模式的架构

基于SSH模式的架构，解决某站机房内作业问题，但随之业务增长SSH并发执行的弊端也显露出来

B站早期实现了一套基于SSH模式提供了一个跨云调度的作业平台。随着业务的扩展，每日百万级别的作业量级，海量作业并发效率，编排化的流式作业，我们早期的架构在性能、稳定性和可观测性等方面都无法支撑

21年开始，我们利用Kratos框架快速进行重构迭代，架构升级，Kratos框架组件的插件化，我们在快速迭代的同时可以灵活选择适合的工具链，如公司内部的服务注册发现，orm/mq/cache，公司内部的可观测组件等，实现快速的问题定位和排障

B站作业平台Job介绍

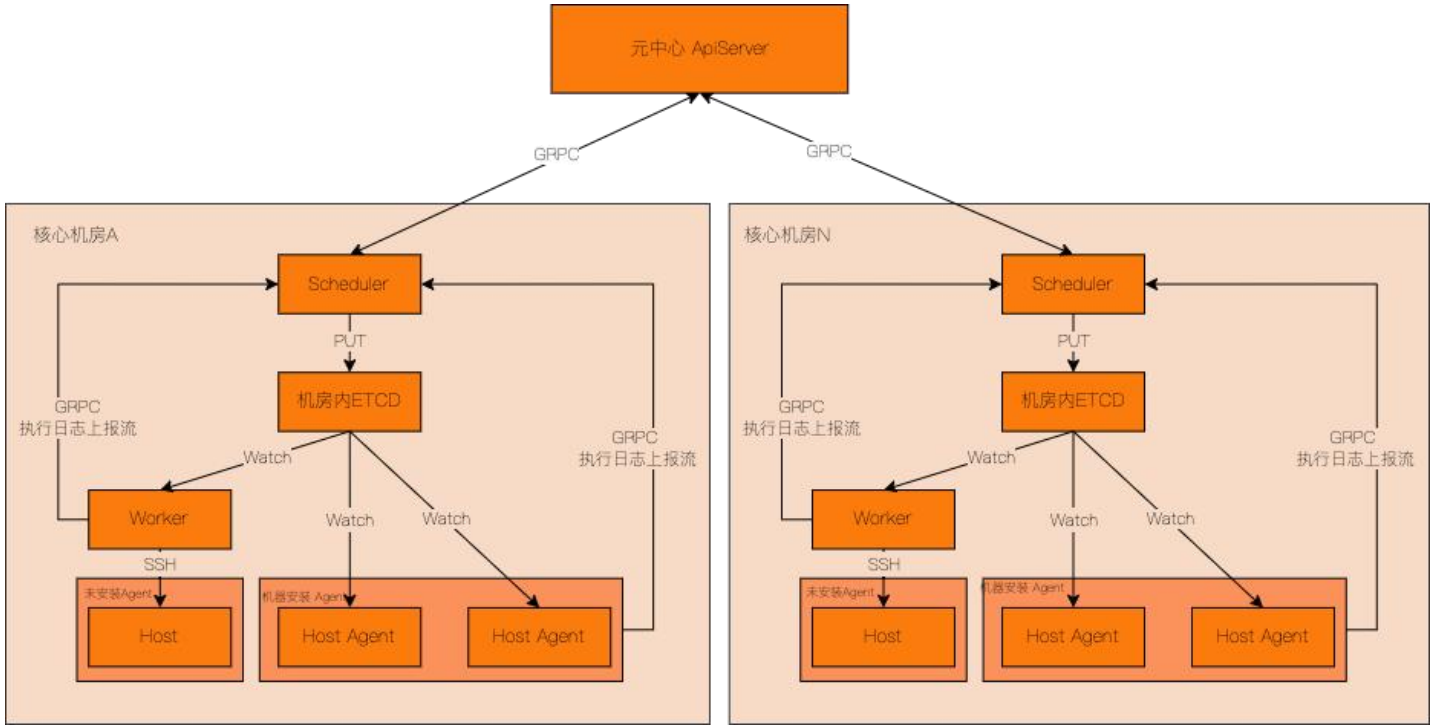


参照K8S整体架构设计，按照作业的生命周期管理划分服务组件: Apiserver, Etcd, Schedueler, Worker四大组件。

- Apiserver 作业平台的大脑，负责统一鉴权，作业任务的初始化等，对外暴露接口提供上层业务调用
- Etcd 作业分发所依赖的中间件，作业存储
- Scheduler作业调度器，主要用来处理作业的调度至Worker的逻辑
- Worker负责作业执行和执行日志上报，职业执行通过调用SSH模块远程执行命令，并把执行STDOUT采集进行上报

缺点：SSH模块并发处理的能力满足不了SRE海量作业并发，长时作业，大作业等场景

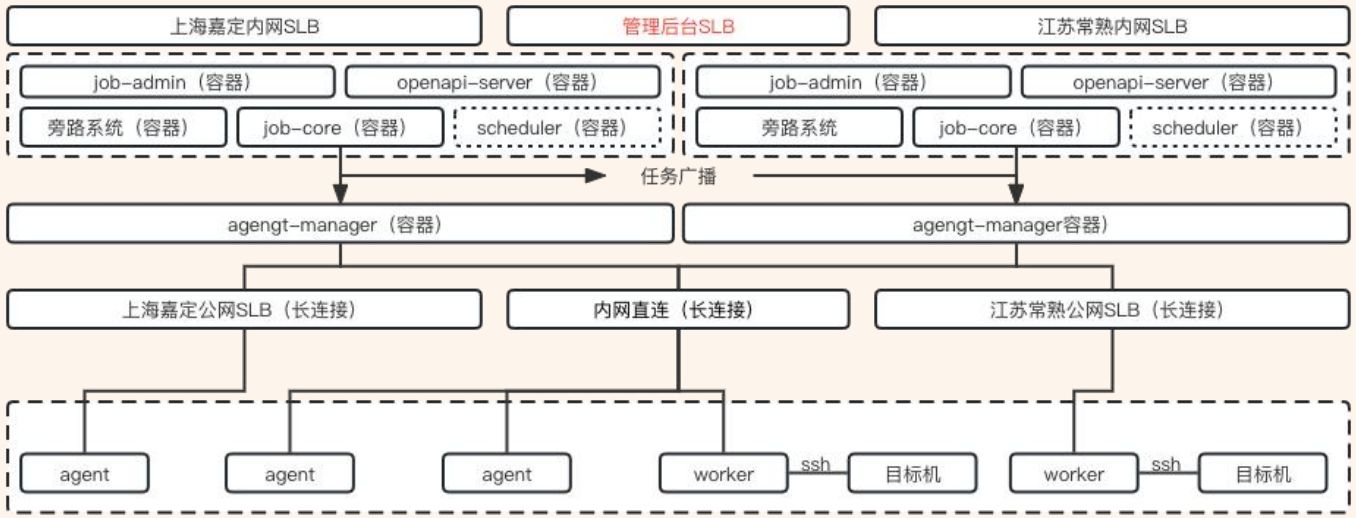
B站作业平台Job介绍



引入Agent解决Woker组件并发处理时的性能、稳定性问题。迭代期间也解决了如下问题：

- Agent初始化，通过ansible或作业平台SSH的功能完成Agent的安装和初始化
- Agent心跳，故障掉线的恢复工作
- Agent侧作业量，连接数，机器负载等监控指标采集
- 拆分Etcd集群，按照Regin划分，减轻Etcd压力

B站作业平台Job介绍



- job-admin 作业辅助/边缘/内部功能接口与实现
- openapi-server 作业对外开放的open接口
- job-core 作业核心功能实现
- agent-manager agent/worker管理与作业任务处理
- worker ssh执行代理
- agent 物理机执行代理
- event旁路 目前核心作用日志消费并持久化

多活场景下保证单机房作业原子执行

- 正常情况下机房A对外提供服务
- 单机房不可用后，上层切流至机房B，机房B开始增量作业调度
- 异常期间不支持存量作业failover到其他机房调度

第四部分

Job设计实现：Agent/Worker作业执行和上报

为了解决跨越多个基础设施的兼容性问题，我们引入了Agent/Wokrer两大核心组件

- Agent是实现主机与Job作业通讯的专用程序，在主机上安装了作业Agent以后，我们可以通过Job平台对主机进行管控，包含作业执行，文件分发，数据上报，基础信息采集等
- Worker是基于SSH协议实现远程作业执行的程序。对调度到本节点的作业/文件任务进行执行和日志上报

Agent Executor定义

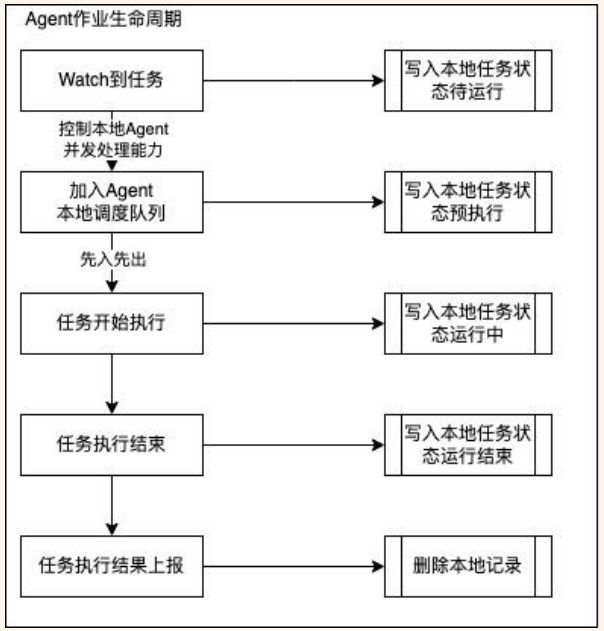
```go
// Executor 执行器接口
type Executor interface {
    // ExecFileUploadTask 执行文件上传任务
    ExecFileUploadTask(ctx context.Context, task *global.AgentTask)
    // ExecNormalTask 执行普通任务
    ExecNormalTask(ctx context.Context, task *global.AgentTask)
    // FinalLogQueuePop 弹出最终日志
    FinalLogQueuePop() interface{}
    // InstantLogQueuePop 弹出实时日志
    InstantLogQueuePop() interface{}
    // KillTaskByReq 杀死任务
    KillTaskByReq(task *global.AgentTask)
    // InstantLogQueuePush 写入实时日志
    InstantLogQueuePush(ctx context.Context, value interface{}) (cancel bool)
    // SetOneSelf 独立节点模式
    SetOneSelf()
    // RecoverTask 恢复任务
    RecoverTask() (err error)
}
```

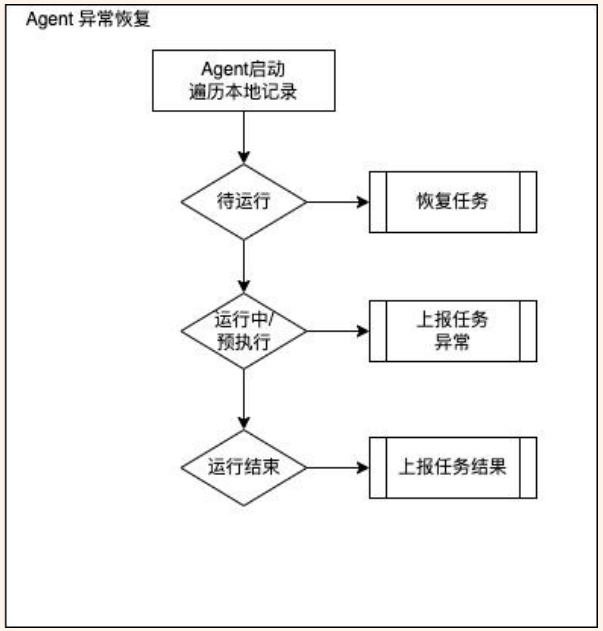
- ExecFileUploadTask：负责执行文件上传任务
- **ExecNormalTask：负责快速执行**
- FinalLogQueuePop：负责最终执行日志上报
- InstantLogQueuePop：负责实时日志输出
- KillTaskByReq: 中止任务
- **InstantLogQueuePush: 推送实时日志**
- SetOneSelf: 独立节点模式
- **RecoverTask: 恢复任务**

Agent 作业生命周期

```go
type dao struct {
    db    *sql.DB
    redis *redis.Redis
    mc    *memcache.Memcache
    cache *fanout.Fanout
    etcd  *clientv3.Client
    Js    util.JsonStore
}
```

```go
type JsonStore interface {
    // SaveSJSON 存储已经解析的JSON
    SaveSJSON(json string, subPath string) (err error)
    // SaveJSONByStruct 解析struct并存储JSON
    SaveJSONByStruct(obj interface{}, subPath string) (err error)
    // ReadJSON 读取JSON并返回string
    ReadJSON(subPath string) (result string, err error)
    // LoadStructByJSON 根据JSON 加载一个struct
    LoadStructByJSON(obj interface{}, subPath string) (err error)

    LoadStringsByRoot() (files []string, err error)
}

type jStore struct {
    RootPath    string `json:"root_path"` // 根路径
    *sync.Mutex `json:"-"`
}

// NewJsonStore 初始json缓存
func NewJsonStore(root string) (js JsonStore) {
    js = &jStore{
        RootPath: root,
        Mutex:    &sync.Mutex{},
    }
    return
}

// SaveSJSON 存储已经解析的JSON
func (j *jStore) SaveSJSON(json string, subPath string) (err error) {
    j.Lock()
    defer j.Unlock()
    return WriteStringToFile(JoinPath(j.RootPath, subPath), json, perm: 0644)
}
```

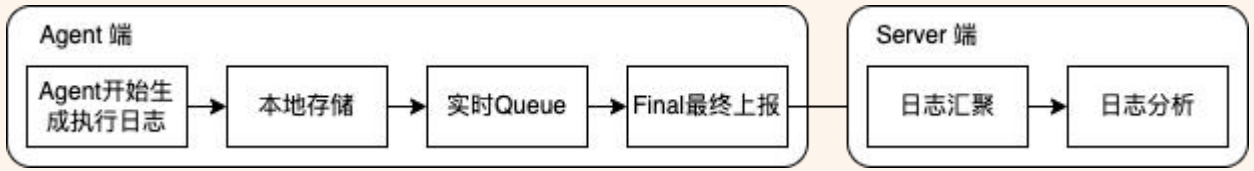




- **本地记录任务状态**
  - Agent异常终止后可以试图恢复待执行的任务
  - 增加运维排查问题方向，便于快速定位问题
- **Agent异常恢复**
  - 读取本地记录任务根据任务状态进行处理

Agent 快速执行 – 日志上报

```go
func (e *executor) runShellTask(ctx context.Context, task *model.Task) {
    if task.User == "" { // 设置执行用户
        task.User = "root"
    }
    user, err := user.Lookup(task.User)
    if err != nil {...}
    uid, _ := strconv.Atoi(user.Uid)
    gid, _ := strconv.Atoi(user.Gid)
    log.Info(fmt.Sprintf("Uid #{uid}  Gid #{gid}"))
    task.Cmd.SysProcAttr = &syscall.SysProcAttr{Setpgid: true}
    task.Cmd.SysProcAttr.Credential = &syscall.Credential{Uid: uint32(uid), Gid: uint32(gid)}
    log.Info( format: "set req_user:%s user:%s uid:%d gid:%d", task.User, user.Name, uid, gid)
    stdout, err := task.Cmd.StdoutPipe()
    if err != nil {...}
    task.Cmd.Stderr = task.Cmd.Stdout // 合并两种std输出
    if err = task.Cmd.Start(); err != nil {...}
    go e.killProcessWithTimeOut(task, task.TimeOut)
    reader := bufio.NewReader(stdout)

    for status := true; status; {
        line, err := reader.ReadString( delim: '\n')
        switch err {
        case io.EOF: // 表示脚本退出
            status = false
        case nil: // 表示脚本正常运行
            log.Info( format: "shell task(%d)执行日志: %s", task.Id, line)
            task.InstantLog <- line
            task.AddFinalLog(line)
        default: // 脚本出现任何异常
            log.Warn( format: "脚本非自然终止: %v", err)
            status = false

        }
    }
    err = task.Cmd.Wait()
```

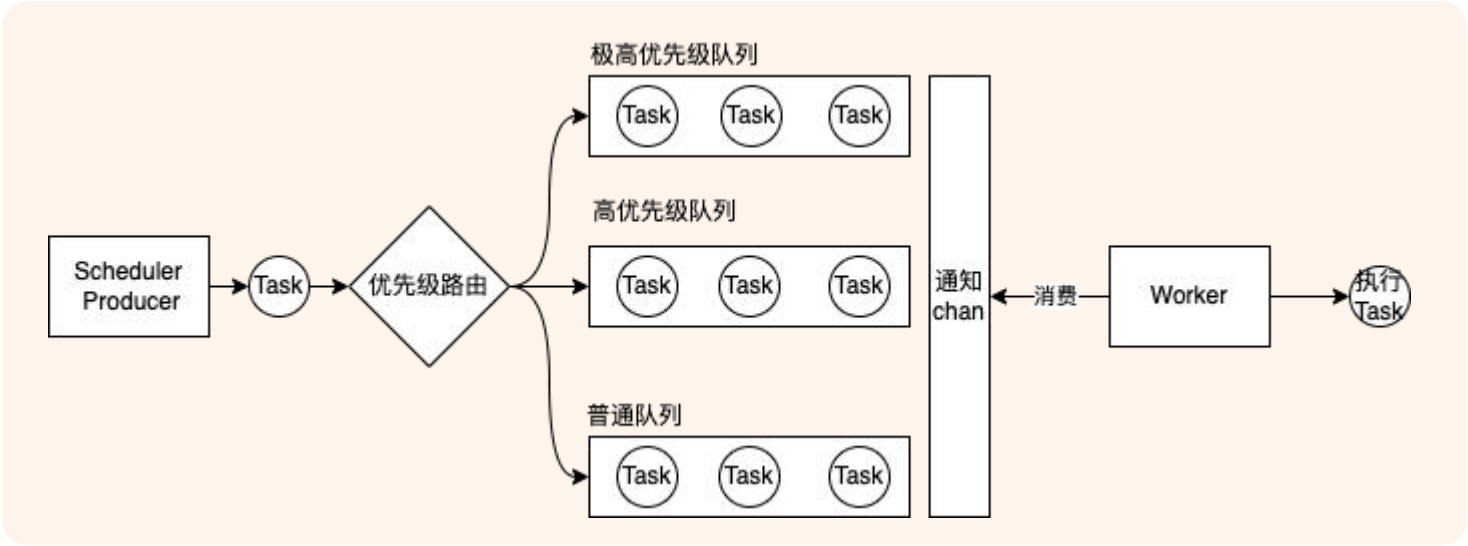


- 实时日志上报失败不补偿，由最终日志上报保证执行日志上报是否成功
- 重试失败日志上报，防止同一时刻大量请求，采用斐波那契数列规避控制等待时间
- 日志到Server端后，采用公司统一日志框架进行日志汇聚分析

Worker引入三种优先级队列

```go
// Queues ...
type Queues struct {
    TaskResultQueue  queue.Queue // 任务结果处理队列
    TaskPool         queue.Pool  // 作业资源池，控制并发任务
    FPQueue          queue.Queue // 最优先的任务队列
    GPQueue          queue.Queue // 优先的任务队列
    NPQueue          queue.Queue // 通常的任务队列
    FileDownloadPool queue.Pool  // 文件下载资源池，控制并发任务
}
```

```go
// handlePriorityTasks handle task from task queue
func (s *Service) handlePriorityTasks() {
    defer func() {...}()
    pn := int(s.cfg.TaskConfig.PriorityNum)
    for {
        s.handleFirstPriorityTasks(pn * 3)
        s.handleGivePriorityTasks(pn * 2)
        s.handleNotPriorityTasks(pn)
        time.Sleep(time.Millisecond)
    }
}
```



- 初始化默认三种优先级队列，由Scheduler根据业务特性进行调度
- handlePriorityTasks负责消费三种优先级队列中的task进入taskChan，然后执行
- 防止高优先级的队列一直不断插入，导致低优先级task饿死，增加权重随机数
    - 高优先级队列更大概率被消费，低优先级概率较小

Worker定义

```go
type Worker struct {
    WorkerId  string         `json:"worker_id"`
    Heartbeat time.Time      `json:"heartbeat"`
    TaskIds   []string       `json:"task_ids"` // 在当前worker上认领的task列表
    LoadAvg   load.AvgStat   `json:"load_avg"` // 1/5/15 负载情况
    Mem       procfs.Meminfo `json:"mem"`      // 内存情况
    State     bool           `json:"state"`
}
```

```go
// RegisterToETCD 注册
func (s *Service) RegisterToETCD() {
    fib := util.NewFibonacci(util.DefaultInitFibonacciSlice, util.DefaultMaxNumberSize)
    leaseID, heartChan, err := s.KeepOnline()
    if err != nil {
        log.Error( format: "ETCD心跳注册失败 err: %v", err)
        panic(err)
    }
    go func() {...}()
}
```

```go
for _, watchEvent := range watchResp.Events {
    taskEvent := new(global.TaskEvent)
    switch watchEvent.Type {
    case mvccpb.PUT:
        log.Infof( format: "receive new task: %s", watchEvent.Kv.Key)
        taskObj, err := w.unpackTask(watchEvent.Kv.Value)
        if err != nil {
            continue
        }
        taskEvent = w.buildTaskEvent(global.TaskEventInsert, taskObj, watchEvent.Kv.ModRevision)
    case mvccpb.DELETE:
        taskID := filepath.Base(string(watchEvent.Kv.Key))
        taskObj := &global.Task{ID: taskID}
        log.Infof( format: "receive delete task: %s", taskID)
        taskEvent = w.buildTaskEvent(global.TaskEventDELETE, taskObj, watchEvent.Kv.ModRevision)
    }
```

Worker:

- **注册并定时上报心跳到ETCD**
- **watch etcd ‘/job/worker/{worker_id}‘:**
  - **PUT event:**
    - task status == "scheduled" ： 更新worker 的 taskIds 列表
    - task status == "stopped"：从队列中移除， 更新worker的 taskIds列表
  - **DELETE event:**
    - task status == "scheduled"：从队列中移除， 更新worker 的 taskIds 列表 ， workers 的 taskids 列表 加分布式锁 [WorkersLock]

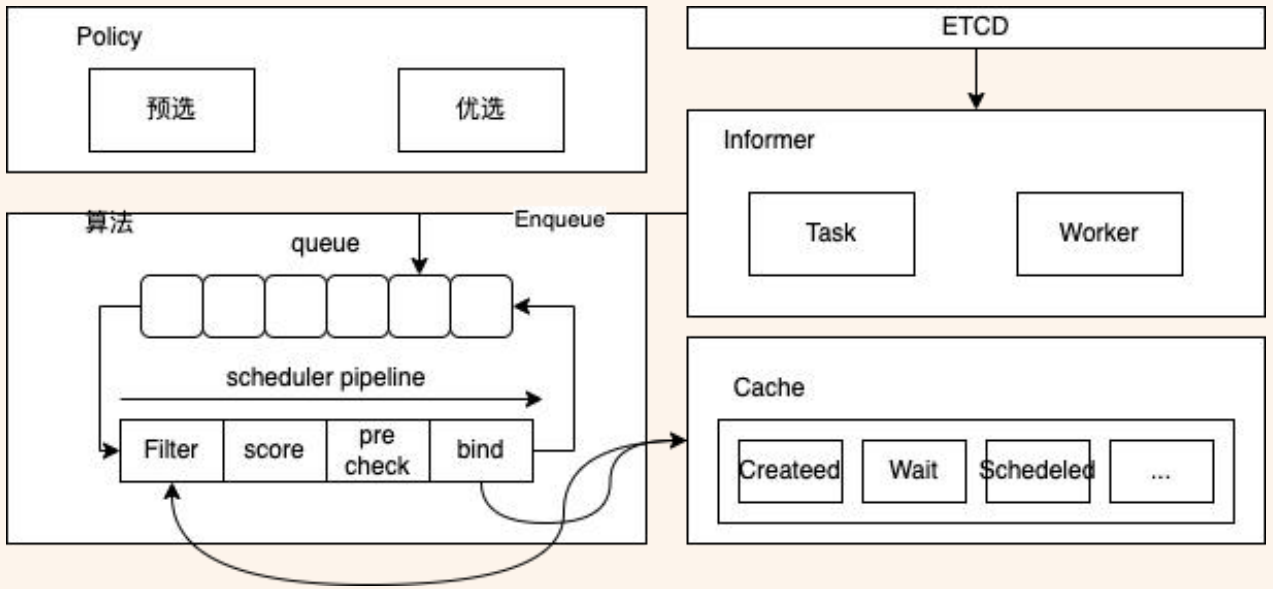
第五部分

Job设计实现：Scheduler作业调度

为了解决多云厂商间作业的调度和集成问题，我们引入Scheduler组件，同时加入逻辑可用区的概念Regin+Zone

Scheduler作业调度器，主要用来处理作业的调度至Worker的逻辑，由于B站机房的复杂性以及部分作业的重要及时性，Scheduler也是实现了部分算法

Scheduler调度架构



Policy：

- 策略模块管理Scheduler的调度策略，实现比较简单仅支持预选和优选，不扩展

Informer：

- 在服务启动时，调度器会注册Informer来监听Task/Worker的变化

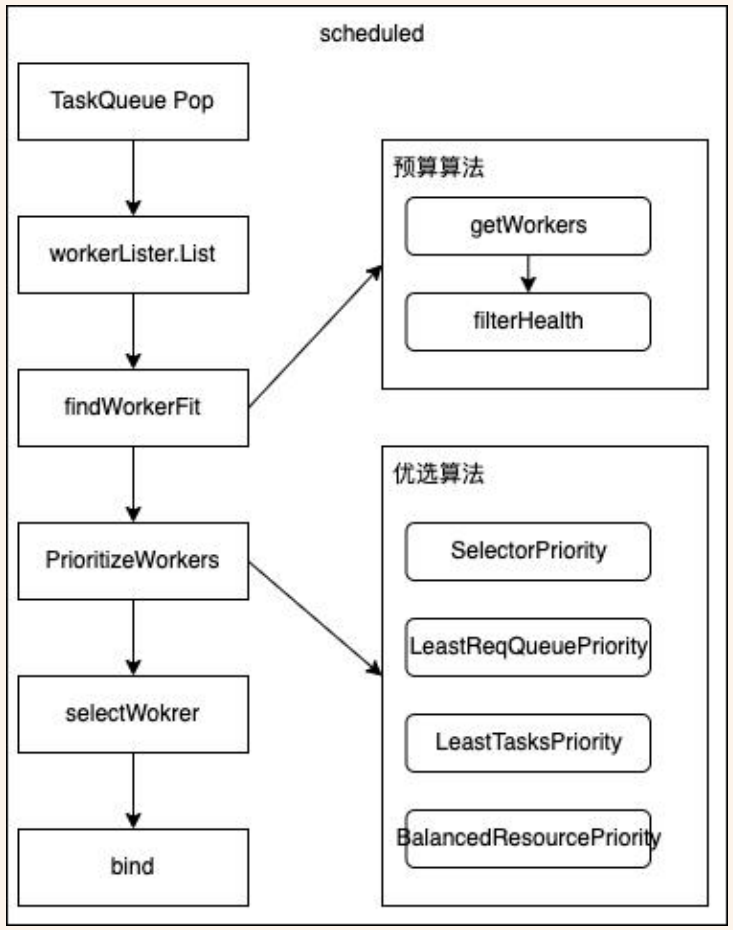
Schedule Cahe：

- Schedule Cache用于缓存Informer更新的资源信息和调度的结果信息，以便调度的时候可以进行快速的查询

算法

- 负责实现调度器的核心作业调度逻辑，整体上来讲分为四个阶段：Filter阶段选择适合Task的Worker，Score阶段对所有符合条件的Worker进行打分，PreCheck阶段校验Worker，Bind阶段下发任务更新状态

Scheduler Algorithm



Scheduler默认简单实现了两类调度算法

- 预选调度算法：获取符合运行“待调度Task资源对象”条件的workers
- 优选调度算法：为每一个可用节点计算出一个最终分数，择优选择Worker

最终得分计算公式

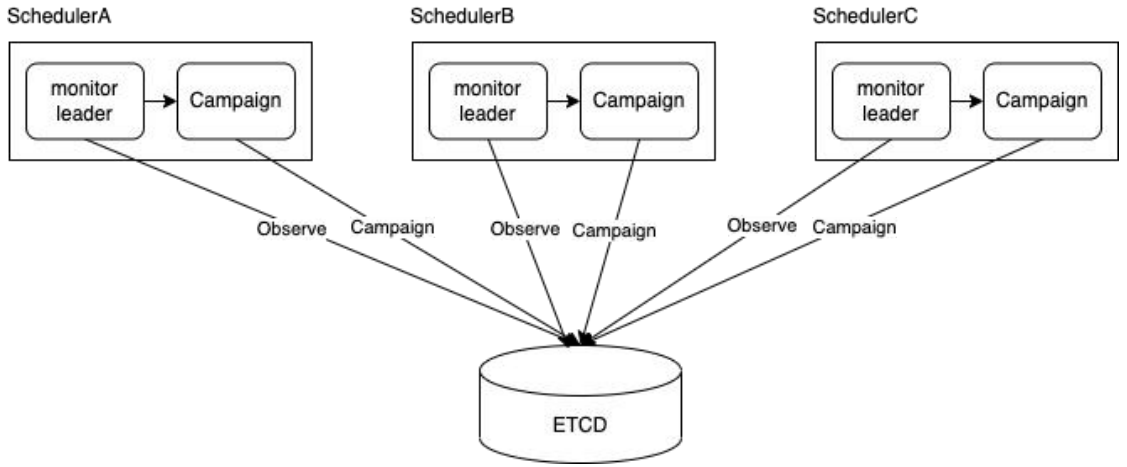
finalScoreWorker1 = (weight1*priorityFunc1)+(....)+(weightn*priorityFuncn)

在计算完所有节点的分数后，Scheduler 选择得分最高的Worker作为运行当前Task的节点。如果预选筛选只有唯一节点满足跳过优选阶段，如果多个节点同时最高分，则随机选择

优选调度算法	说明	默认权重值
SelectorPriority	尽量不跨 region/zone 调度作业	1
LeastReqQueuePriority	计算作业所需优先级队列，当前worker阻塞情况，队列长度越小worker越优	1
LeastTasksPriority	计算当前worker的作业队列长度，越小越优	1
BalancedResoucePriority	计算worker上 CPU 和内存的使用率，使用率最均衡的wokrer最优	1

Scheduler领导者选举

```go
// Run ...
func (s *Scheduler) Run() {
    go s.election()
    for range time.Tick(1 * time.Second) {
        select {
        // 选举成功:  开启调度服务, watch etcd wait task & failover
        case <-s.electionCh:
            // 更新apiserver当前活跃scheduler
            go service.Svc.SetActiveScheduler(s.ID)
            // 监听新任务
            go s.watchTasks( fromCurrent: false)
            // 为新启动的任务分配调度
            go s.handleTaskEvent()
            // 遍历 workers collections, 若有 worker 故障,  做 failover
            go s.watchWorkers( fromCurrent: false)
        }
    }
}

func (s *Scheduler) election() {  1个用法
    log.Info( args...: "start election scheduler")
    ctx, cancel := context.WithCancel(s.Ctx)
    defer cancel()
    for range time.Tick(time.Duration(config.Conf.Sched.ElectionInterval) * time.Second) {
        session, err := concurrency.NewSession(s.EtcdClient.Cli, concurrency.WithContext(ctx), concurrency.WithTTL( ttl: 2))
        if err != nil {
            log.Errorf( format: "start scheduler new session err : %#v, retry after 5s", err)
            continue
        }

        elec := concurrency.NewElection(session, global.EtcdSchedulerElection)
        if err := elec.Campaign(ctx, s.Name); err != nil {...}

        // 选举成功,传递信号到channel
        s.electionCh <- struct{}{}

        log.Infof( format: "scheduler %s win the election", s.ID)
    }
```



我们将每个Scheudler内的主节点管理分成两部分，选举部分和监听部分。

- 选举部分负责选举
- 监听部分负责主备节点切换

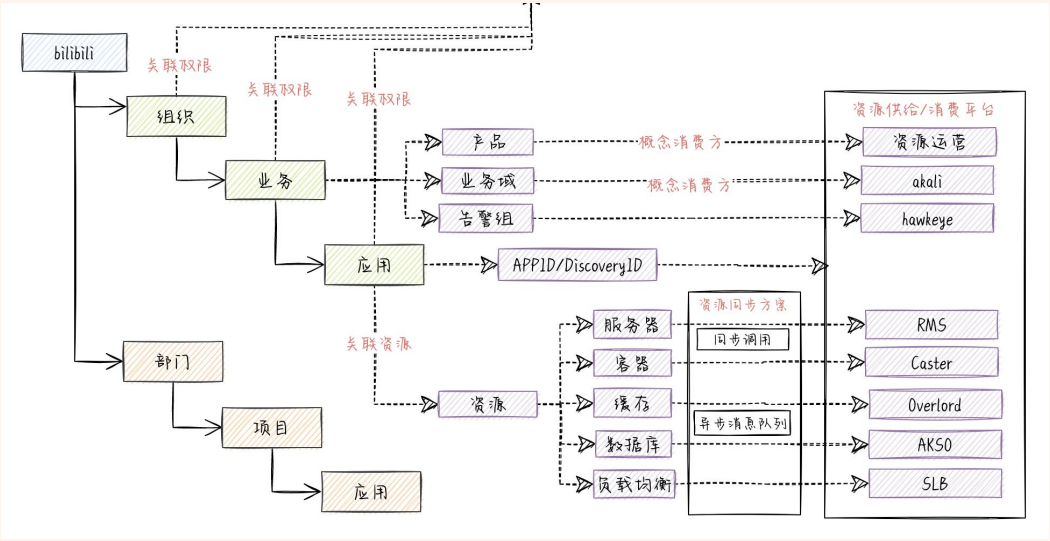
第六部分

Job设计实现：ApiServer鉴权+数据处理

我们设计ApiServer组件，基于CMDB提供的资源管理和统一资源鉴权能力，解决资源权限管控的问题

Apiserver 作为作业平台的大脑，除了负责统一鉴权，还负责作业任务的初始化, 作业状态流转等，对外暴露接口提供上层业务调用

作业的优先级如何确定

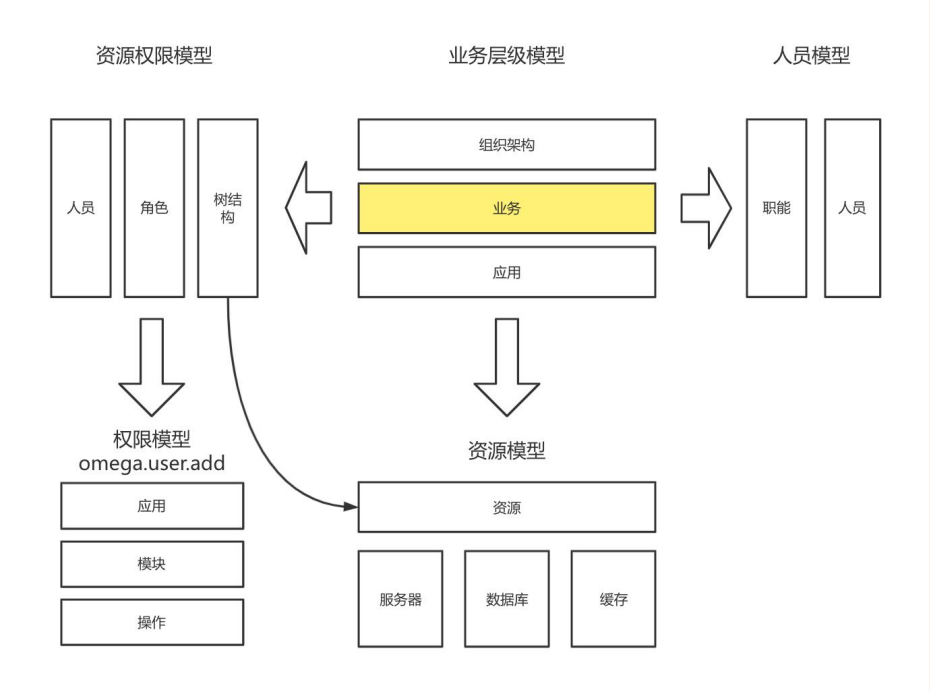


作业平台支持分两种方式支撑作业的优先级

- **基于CMDB业务/应用等级确定优先级**
  - 对接内部CMDB平台，基于CMDB以应用为中心的设计理念，依托组织.业务.应用这样的树形结构进行资源的管理。业务/应用由SRE指定相应的等级，它既是故障时人工保障的等级，也是业务上核心重要程度，更是任务/作业的调度和容器资源优先级。
- **基于场景维度确定优先级**
  - OpenApi/SDK接入时，梳理业务场景，业务SLO

作业平台和CMDB紧密结合

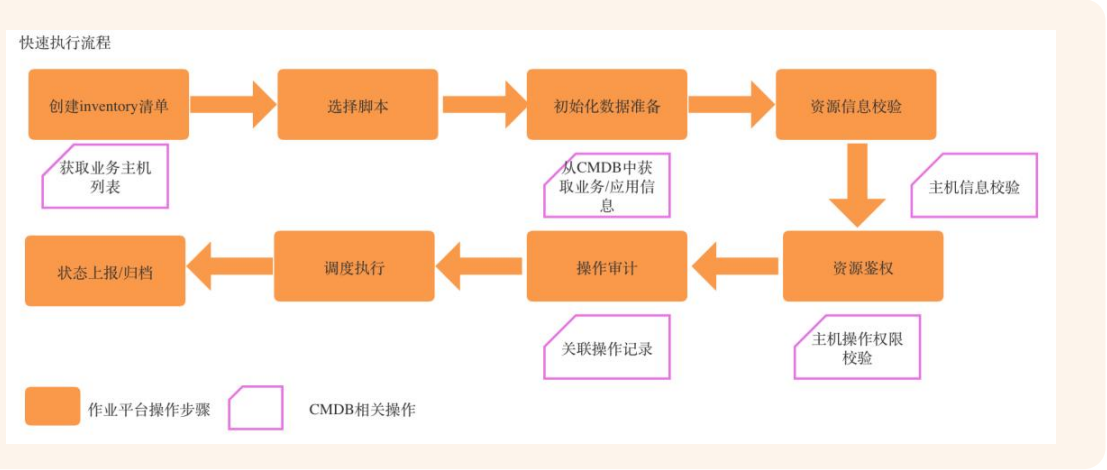
作业的权限如何控制

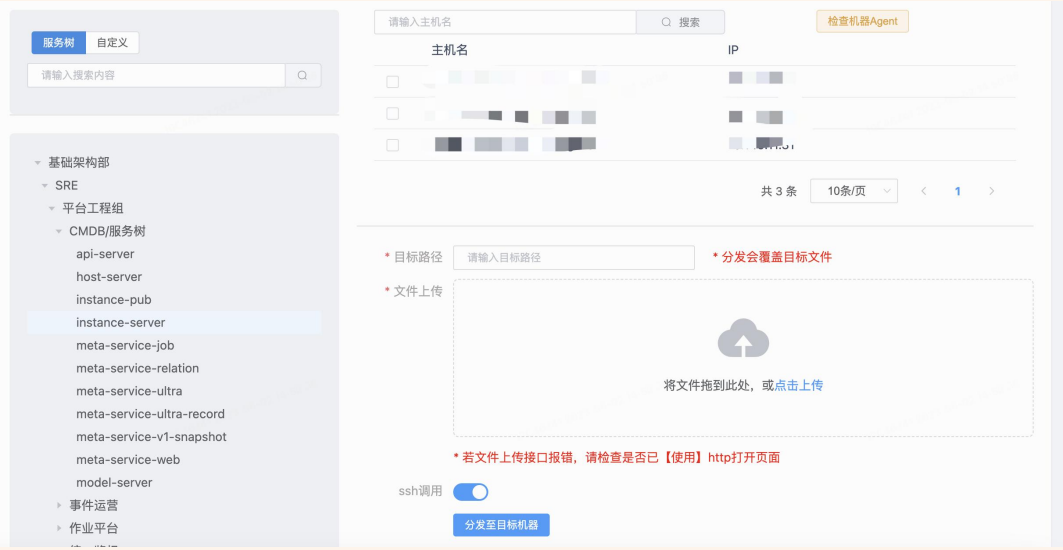


CMDB鉴权复用了IAM的鉴权体系（参照了GCP IAM的实现），主要包含四个部分：

- **成员（member）**
  - 员工账号（OA）
  - 服务帐号
- **角色（role）**
  - 一个角色对应一组权限
  - 权限决定了可以对资源执行的操作
  - 向成员授予某角色，即授予该角色所包含的所有权限
- **权限（privilege）**
  - 权限决定了可对资源执行的操作，由资源提供方接入IAM的时候预先定义，以 <provider>.<resource>.<verb> 的形式表示，例如 galaxy.node.create ，权限不能直接赋予成员，而是需要绑定到角色，再将角色授予成员
- **授权（authorization）**
  - 将一个或多个成员绑定到某个角色称之为授权
  - 某个节点的授权定义了谁 （成员）对该节点的资源拥有何种访问权限（角色）

以快速执行为例简单介绍Server端处理作业流程





Task创建启动流程：

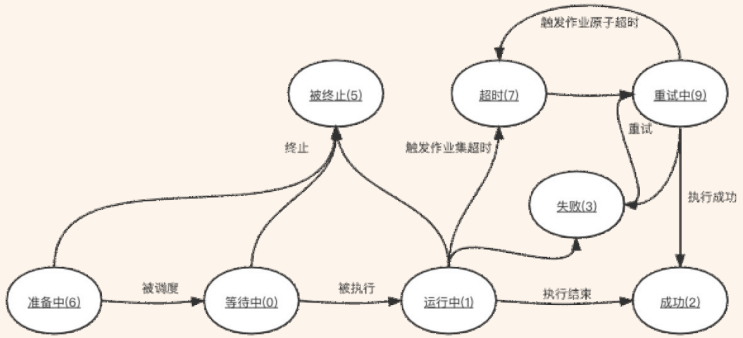
添加task：

1. CMDB资源信息校验，完善作业等级
2. PUT到ETCD中，task的status为created，workerId为空
3. add task to DB，task的status为created，workerId为空
4. 添加用户权限
5. 记录操作记录
6. 返回 taskInfo

启动task：

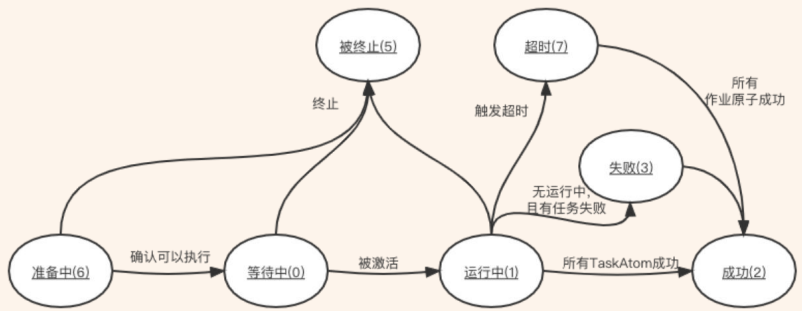
1. 验证task的status 为 stopped 或 created， 否则提示当前状态，且无法启动
2. 加分布式锁，ttl=10，retry=3,失败则提示稍后重
3. PUT 到 etcd，该 taskId 对应的 status 为 wait, 等待scheduler调度
4. 保存到db，status 更新为 wait, 等待scheduler调度
5. 记录操作记录
6. 解锁

作业的状态流转



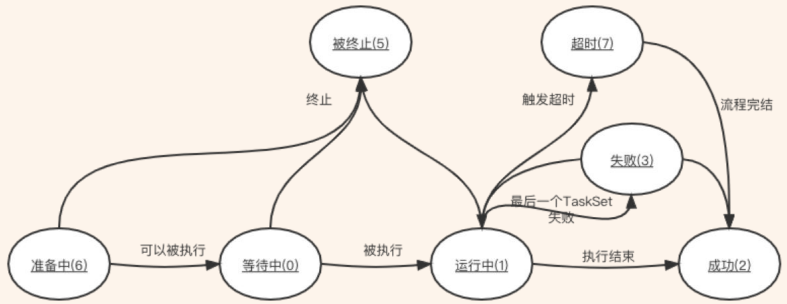
作业原子状态流转如左侧图所示

- 作业原子超时会记录状态同时触发作业集的超时处理
- 作业原子的状态流转由自身控制，不被继承



作业集状态流转如左侧图所示

- 作业的状态流转依据作业原子
- 超时重试时，作业集至作用在该集的运行中的作业原子
- 设置了并发度的，需要按规则配置作业集和作业原子的超时



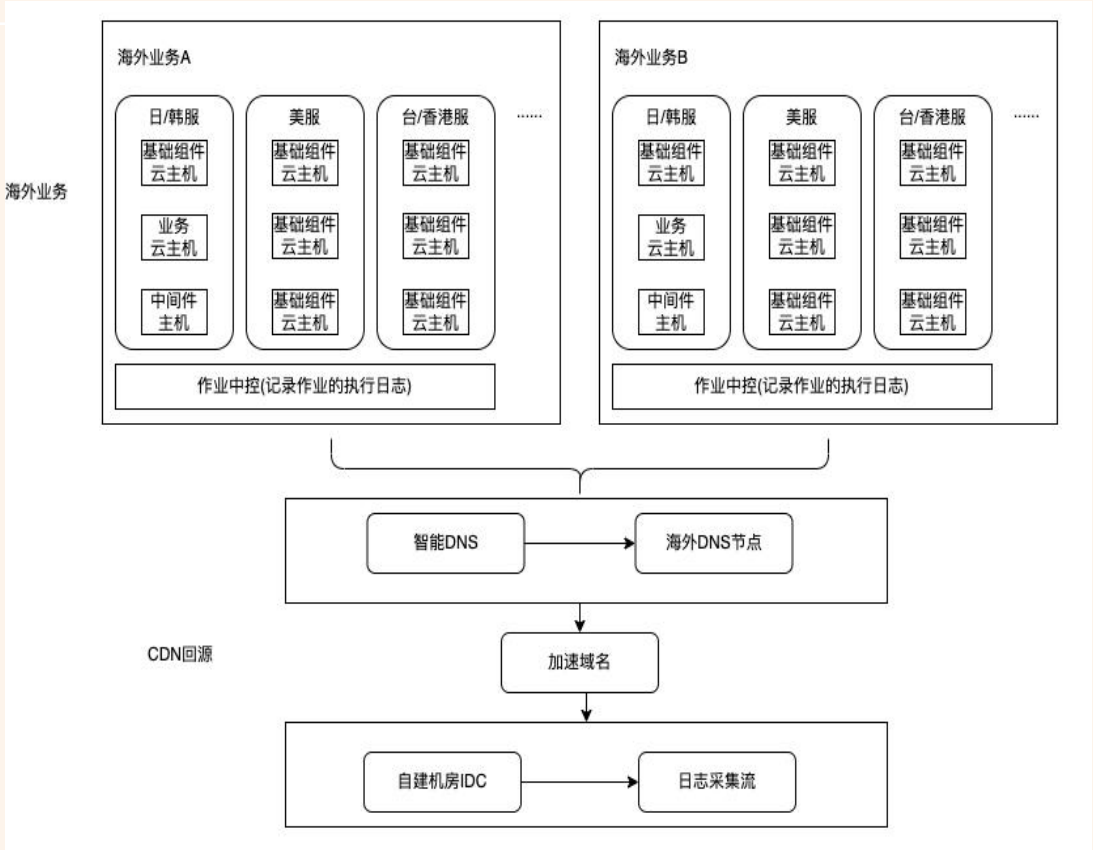
作业流状态流转如左侧图所示

- 作业流状态流转依据作业集
- 作业流全局的设置超时时长按照实际作业集的执行耗时递减，最终判断是否超时

第七部分

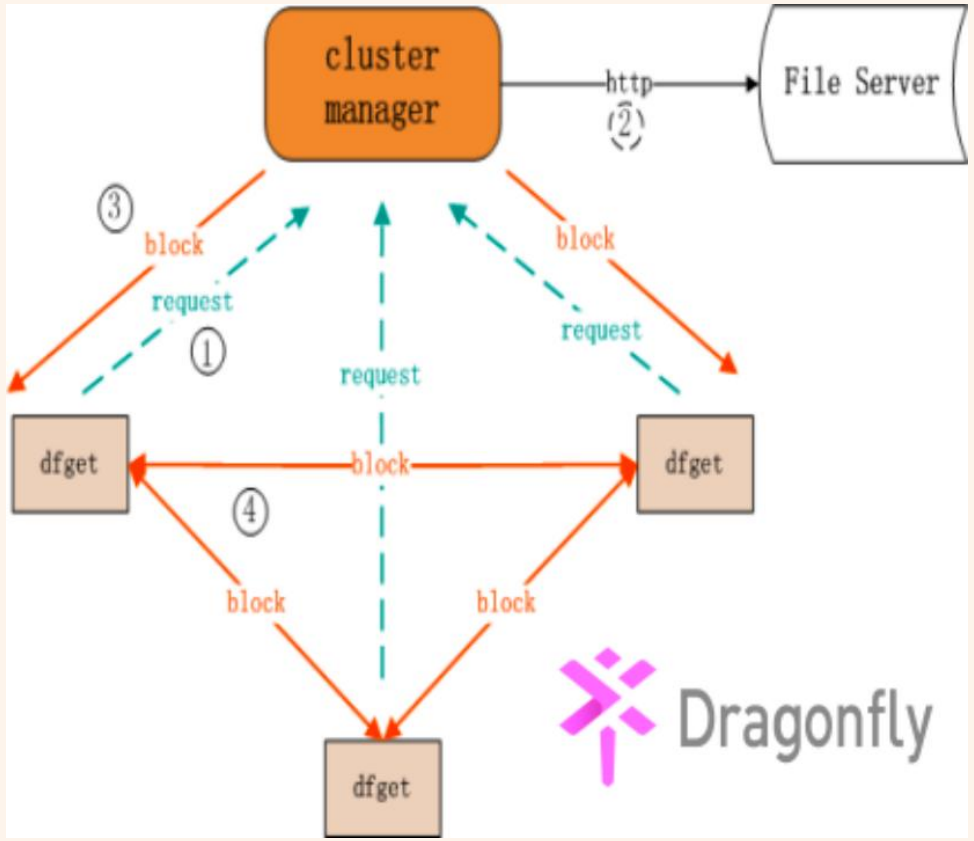
Job设计实现：其他技术难点和细节

海外日志回源



- 海外作业的执行日志保存在海外作业中控机上
- 中控机日志走海外CDN回国内中心机房
- 整个日志流依托公司内部日志采集实时流式传输
- 传输过程无法保证数据100%不丢失，中控机日志需要保存7天内，定时清理

文件分发



需求背景：

使用Sftp进行文件传输，对于大批量文件处理性能一直存在问题，海外云主机之间的网络延时经常导致传输超时

解决方案

- 采用Dragonfly v2版本集成

降低源站压力，保障源站稳定性和响应速度，海外文件分发效率提升90%

- Agent集成dfget功能

dfget相关源码改造，集成在作业Agent中

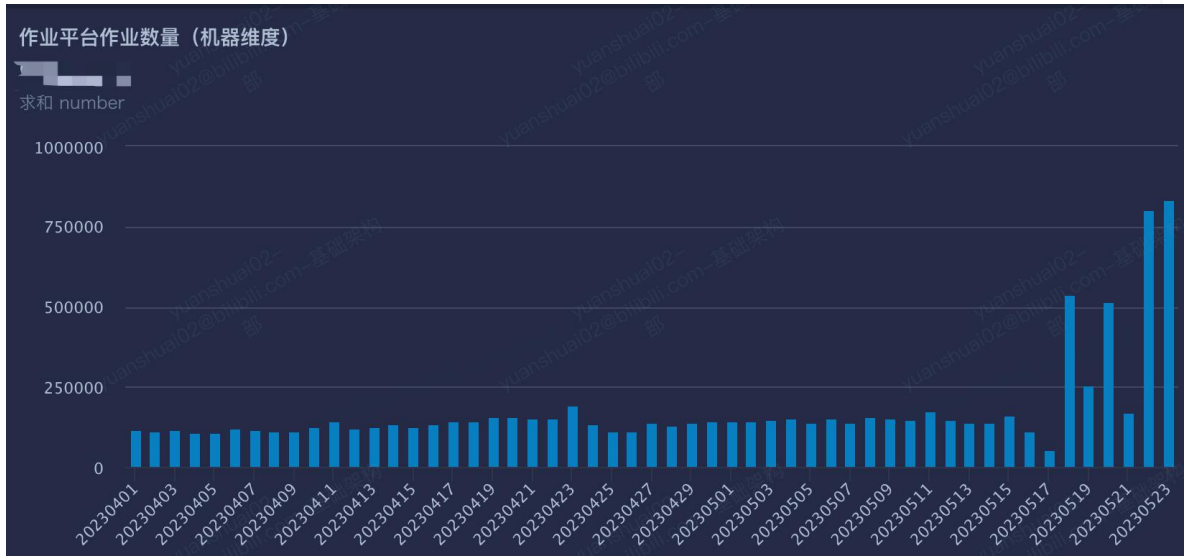
- IDC和公有云云上隔离部署

第一部分

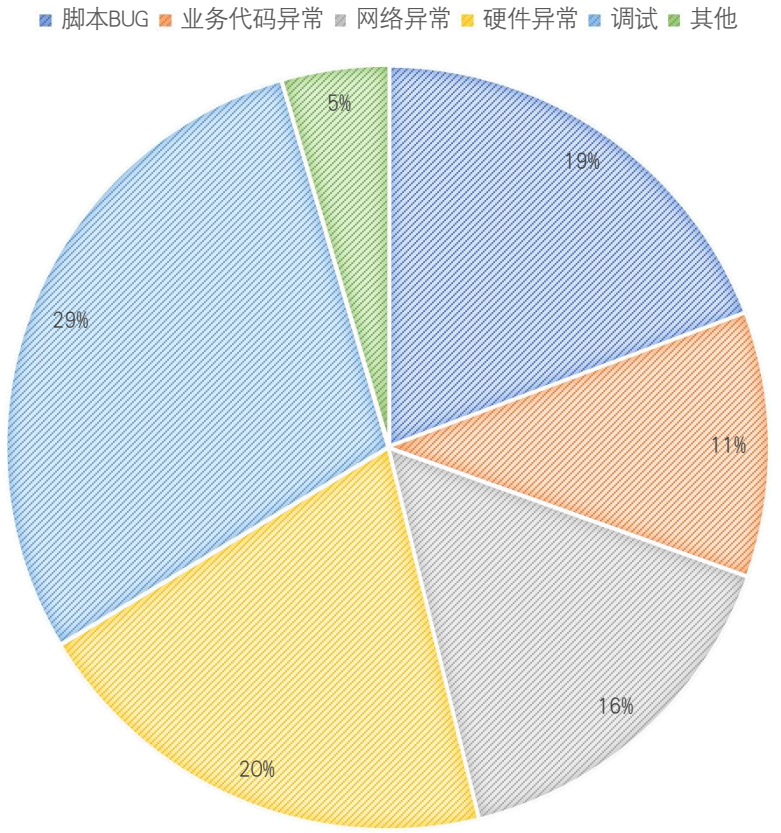
总结展望

运维可视化

作业平台执行的数据，辅助SRE/研发同学查看
历史执行的次数和执行效率



作业运行异常的分析，集中关注处理异常比例高



不，你好奇

上海 杨浦

扫一扫上面的二维码图案，加我为朋友。

哔哩哔哩技术

微信扫描二维码，关注我的公众号